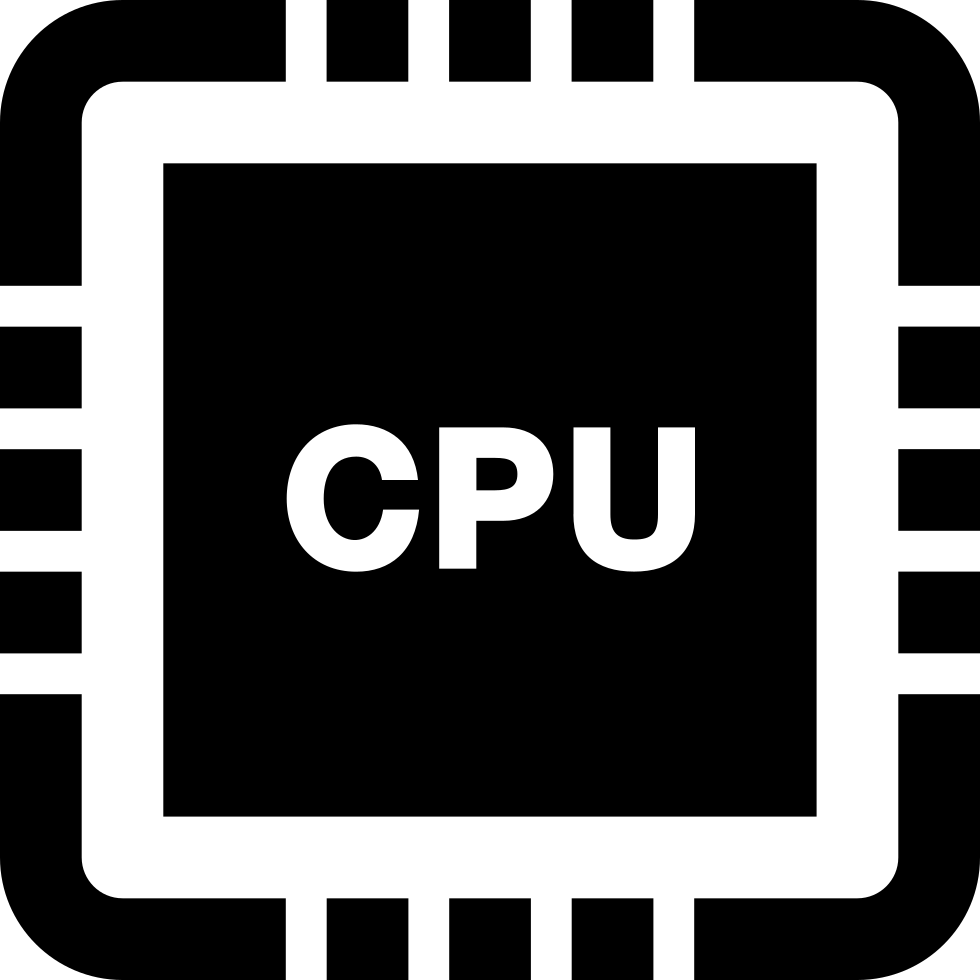
**Authors:**

Blake Fullerton

Rusty Dillard



***CS433: Assignment 1 Report***

Submitted Files:

* pcb.h
* pcbtable.h
* pcbtable.cpp
* readyqueue.h
* readyqueue.cpp

***How to Compile/Run the Program:***

*To compile, command used: make*

*To run, command used:*

* *./test1 && ./test2*

**1 Overview of Program**

**1.1 Results and runtime of test 2**:

Our program ran both test 1 and 2 with all the supported files in complete fashion. We ran the tests multiple times to make sure that there were not any outliers or funky bugs that were skewing our data. With zero errors and a very low standard deviation we know that our code is reliable and consistant.

**1.2 Choice of Data Structure:**

The data structure chosen initially to complete the assignment fully without extra credit was just a straightforward array. There was no sorting to it; the only algorithm was to find the index of the highest priority in the ready queue and return that PCB pointer. That implementation was good enough to run test01 successfully for full credit, however, it wasn’t fast enough while running test02. We went ahead and submitted the code of what we had and decided to come back to the ready queue assignment if there was time at the end of the week to complete the extra credit.

In order to complete test02, a new data structure was chosen: the maxheap. Rusty had notes from his data structures class and was able to translate it into an implementation of a ready queue. The reason this algorithm and structure was chosen is because of its speedy sorting and selection abilities. The maxheap keeps the highest priority PCB at index 0, using sifting algorithms, so the time complexity of the search for the highest priority PCB turns out to be O(1). Time complexity from sorting using the siftUp and siftDown functions is O(nlog(n)). This is much faster than other sorting algorithms like bubble sort and selection sort whose time complexities are O(n^2) due to the number of comparisons they need to make. The next section will detail the implementation of the maxheap in the ready queue project.

**1.3 Files and What they Do:**

1. **pcb.h:** Process Control Block header file that contains the data members and functions that allow operation on and access to the PCB objects that will be created for other classes in other files. The PCBTable object will store generated pointers to new instances of the PCB object instantiated in the heap and feed them to the ReadyQueue object. The following are the data members and functions included in pcb.h:
   1. Data members:
      1. unsigned int id
      2. unsigned int priority
      3. ProcState state
   2. Class functions:
      1. Overloaded constructor with the following parameters:
         1. unsigned int id
            1. Initialized to 0
         2. unsigned int priority
            1. Initialized to 1
         3. ProcState state
            1. Initialized to ProcState::NEW (an enumerator)
      2. Destructor
      3. getID

- Used to return the PCB object’s ID data member

* + 1. getPriority

- Used to return the PCB object’s priority data member

* + 1. getState

- Used to return the PCB object’s state data member

* + 1. setState

- Used to set the state data member’s value

* + 1. setPriority

- Used to set the priority data member’s value

* + 1. display

- Used to show the contents of the PCB object

1. **pcbtable.h and pcbtable.cpp**: pcbtable.h inherits pcb.h to utilize the functions it has and to allow for the instantiation of PCB pointers for storage in its table. pcbtable.cpp inherits pcbtable.h and implements the functions it outlines. Process Control Block table header and C++ files that contain the data members and functions to be implemented in the pcbtable.cpp file. The PCBTable will store the PCB pointers generated in the main functions of test01.cpp and test02.cpp. PCBTable then provides functions to the ReadyQueue for retrieval of the PCB pointers where it can work with the processes. The files pcbtable.h and pcbtable.cpp contain the following data members and functions:
   1. Data members:
      1. int size
      2. int count
      3. PCB\*\* table
   2. Class functions:
      1. Base constructor

- Initializes the data members size, count, and table

- size initialized to 100

- count initialized to 0

- table allocates a PCB pointer table in the heap

* + 1. Overloaded constructor with the following parameter:
       1. Int size

- All of the data members are then initialized

* + 1. Destructor
       1. Loops through the table, releases the PCB pointer memory, and releases the entire table memory
    2. getPCB
       1. Returns the PCB at the prescribed index
    3. addPCB
       1. Check if the table is full and add the PCB accordingly
    4. addNewPCB
       1. Helper function that creates a new instance of a PCB pointer and calls the addPCB function

1. **readyqueue.h and readyqueue.cpp:** readyqueue.h inherits pcb.h in order to utilize its functions and allow for the instantiation of PCB pointers. readyqueue.cpp inherits readyqueue.h to implement the functions it outlines for sorting and PCB selection. Ready Queue header and C++ files that contain all the functions related to the Ready Queue implementation. ReadyQueue is the class that’s implemented in these files and it emulates the way a Ready Queue would be implemented in a simple operating system. The Ready Queue will take processes passed to it from the PCBTable and then choose the highest priority PCB to run and remove from its table. The ReadyQueue class has the following data members and functions:
   1. Data members:
      1. int length
      2. int capacity
      3. PCB\*\* table
   2. Class functions:
      1. Base constructor

- Initializes the data members length, capacity, and table

* + 1. Overloaded constructor with the following parameter:
       1. int capacity

- All of the data members are then initialized

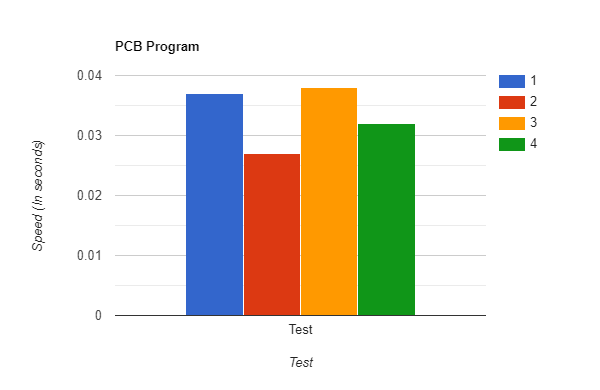
* + 1. Copy constructor
       1. Accepts in a ReadyQueue object as the reference and performs a deep copy of it using a for loop and new pointers for the new PCB table and setting the new ReadyQueue’s data members equal to the referenced object.
    2. Overloaded operator=
       1. Performs a shallow copy from the right side of the operator and stores it in the left side of the operator.
    3. Destructor
       1. All that needs to be done here is to release the memory allocated for the array of pointers; the PCBTable object manages the rest of the releases of the PCB pointers.
    4. addPCB
       1. A PCB pointer variable named pcbptr is passed into this function and its state is set to ProcState::READY. The pointer is then placed at the end of the PCB pointer array table and the length of the ReadyQueue is increased by one.
    5. removePCB
       1. This function removes the highest priority PCB using a for loop to discover the highest priority and also checks if that PCB’s state is set to ProcState::READY. If the state is anything else, the loop just continues on, even if the highest priority PCB has been found.
    6. size
       1. This function is a getter for the class’s length data member
    7. displayAll
       1. Loops through the ReadyQueue object’s table and calls the display function from pcb.h, formatting them nicely with a little space in front of the displayed PCB pointer’s details.
    8. siftUp
       1. Accepts an int argument that it will use to begin a sorting process, moving the PCB pointer higher into the heap.
    9. siftDown
       1. Accepts an int argument that it will use to begin a sorting process, moving the PCB pointer lower into the heap.
    10. parent
        1. The parent index of a value in a heap at index i is calculated as (i - 1) / 2.
    11. left
        1. The left child index of a value in a heap at index i is calculated as i / 2 + 1.
    12. right
        1. The right child index of a value in a heap at index i is calculated as i / 2 + 2.
    13. isLeaf
        1. An index is determined to be a leaf if it is greater than half of the counted number of values in a heap.
    14. swap
        1. Accepts the PCB pointer table and two int values in this case labeled q and i. The PCB pointers in the table at q and i are swapped. This could be done inline wherever needed, but this helps to indicate that swapping occurs at this point and helps keep the code less cluttered.

**2 Efficiency of Program:**

**2.1 Time of Program:**

| **Test** | **Test 1** | **Test 2** | **Test 3** | **Deviation** | **Average** |
| --- | --- | --- | --- | --- | --- |
| **Time (In seconds)** | **0.037** | **0.027** | **0.038** | **σ: 0.00496** | **0.034** |

**2.2 Bar Graph Representation:**

****

**Lessons Learned/Re-Learned:**

*Blake Fullerton:*

* The lessons that I relearned were using vectors and classes. I do not have much experience using multiple classes. It was interesting looking at the test files and seeing what they are calling, and what the output was supposed to look like. By this method I was able to put two and two together by debugging and following the path of what goes where, how, and how much.

*Rusty Dillard:*

* I was able to recall getters, setters, pointers, and data structure and algorithm implementations from 311. I learned how to implement the max heap for a ready queue by basing it off of the priority of each PCB pointer in the ready queue. I learned how a ready queue could choose and run processes from itself using a simple algorithm for finding the highest priority process. I was forced to get used to an IDE I’m not comfortable using, Visual Studio. I typically just use a text editor like Sublime, but I needed something more robust for my debugging.

**3 Final Words and Findings:**

**3.1 References/Resources:**

First starting the assignment we had to sharpen our knowledge on what a PCB really was, for that, we read and went over this wikipedia article to make sure we had everything on point before starting

* [Process Control Blocks Article](https://en.wikipedia.org/wiki/Process_control_block)

Personally, Blake was not super fresh on classes, he used them in CS311 but that long break was spent doing….not studying. Anyways it was nice to have a fresh set of clean code to look at and cross reference calling objects and such

* [CPP Classes](https://www.w3schools.com/cpp/cpp_classes.asp)

**3.1 Conclusions**

In conclusion, the programming assignment on process control blocks helped us to better understand the inner workings of operating systems and how they manage and prioritize processes. We were able to learn about the key components of a process control block, including its state, priority, and resources, and how they are used by the operating system to allocate resources and manage processes. This assignment also allowed us to gain practical experience in implementing these concepts, as well as honing our programming skills. Overall, this assignment was a valuable learning experience that gave us a deeper understanding of the critical role that process control blocks play in the functioning of modern operating systems.

